C14

```
1   /*
2   ** Copyright 1996,1997 EMC Corporation
3   */
5   /*
6   **    EDMDispatchConfig.c
7   **
8   **    Mission Statement:  These are functions that deal with the config file
9   **                        APIs.
10  **
11  **                        Since the config file can't be handled in a
12  **                        thread safe manner we need to mutex lock around
13  **                        all uses of it.
14  **
15  **    Primary Data Acted On:
16  **
17  **    Compile-Time Options:
18  **
19  */
20
21  /*
22  ** The following provides an RCS id in the binary that can be located
23  ** with the what(1) utility.  The intent is to keep this short.
24  */
25  #if !defined(lint)
26  static char  RCS_id [] = "@(#)$RCSfile: EDMDispatchConfig.c,v $ "
27                           "$Revision: 1.23 $ "
28                           "$State: 1997/02/06 20:49:15 $";
29  #endif
30
31  /* #define _POSIX_SOURCE     unable to compile with this define set */
32  /* #define _XOPEN_SOURCE     unable to compile with this define set */
33
34  #include <esl/c_portable.h>
35  #include <esl/ep_xopen.h>
36  #include <esl/inout.h>
37
38  #include <unistd.h>
39  #include <pthread.h>
40  #include <memory.h>
41  #include <sys/time.h>
42  #include <sys/types.h>
43
44  #include <ebconfig/rbconfig.h>
45  #include <EDMDispatchConfig.h>
46
48  static RBC_CONFIGS *rbc = NULL;
49
50  static boolean_ty  first = TRUE;
51
52  static pthread_mutex_t  G_configMtx = PTHREAD_MUTEX_INITIALIZER;
53
54  static void
55  DispatchInitializeConfigMutex()
56  {
57      pthread_mutex_init(&G_configMtx, NULL);
58      first = FALSE;
59  }
```

```
58  void
59  DispatchReadConfig()
60  {
61      eperrno ret;
62
63      if (first == TRUE)
64          DispatchInitializeConfigMutex();
65
66      pthread_mutex_lock(&G_configMtx);
67
68      if (rbc != NULL)
69      {
70          rbc_freeconfig(rbc);
71          rbc = NULL;
72      }
73
74      ret = rbc_parse_config(NULL, &rbc,
75                             RBC_PARSE_DO_NOT_PRESERVE |
76                             RBC_PARSE_APPLY);
78
79      pthread_mutex_unlock(&G_configMtx);
   }
```

```
81    boolean_ty
82    DispatchCheckRestorePermission(char *host, char *username)
83    {
84        int       root = 0777;
85        errr;
86        boolean_ty allowed;

88        // The Configuration structure is not set up correctly !!! STEVE
                                                                        HOWARD

90        pthread_mutex_lock(&G_configMtx);

92        allowed = rbc_canRecover(rbc, host, username, &root, &err);

94        pthread_mutex_unlock(&G_configMtx);

96        allowed=1;
97        return allowed;
98    }
```

```c
/*
**
** Copyright 1996,1997 BMC Corporation
**
** EDMDispatchBackground.c
**
** Mission Statement: This is the entry point for the cleanup thread.
**                    Its main purpose is to do some background processing
**                    that we don't want to do elsewhere.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
** Basic idea here: Module for Background thread.
**
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char   RCS_id [] = "@(#)SCCSfile: EDMDispatchBackground.c,v $
*$Revision: 1.23 $ *
*$Date: 1997/02/06 20:49:15 $ *";
#endif

#define _POSIX_SOURCE       unable to compile with this define set
#define _XOPEN_SOURCE       unable to compile with this define set

#include <restore/dispatch_daemon.h>
#include <EDMDispatchConfig.h>
#include <EDMDispatchSession.h>
#include <EDMTimeMessage.h>
#include <EDMDispatchBackground.h>

#include <unistd.h>
#include <sys/time.h>
#include <pthread.h>
#include <memory.h>
#include <sys/types.h>

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/in.h>
#include <esl/inout.h>

static const int oneDaySeconds = SECONDS_PER_DAY;

// Number of background activities run from the cleanup thread
#define NUMBER_OF_ACTIVITIES 4

static const int configSeconds = 30;   // 30 seconds
static const int oneMinute     = 60;   // one minute in seconds

// Structure used in the cleanup thread to schedule background
//  activities
struct Schedule {
    long frequency;
    long lastrun;
    long nextrun;
    void (*cleanupfunc)();
};
```

```c
void *
DispatchBackground(void *buff)
{
    time_t    curTime;
    time_t    sleepfor;
    time_t    difference = 0;

    // These are all the activities that are scheduled
    struct Schedule   sched[NUMBER_OF_ACTIVITIES] = {
    // Frequency,        lastrun, nextrun, function to call
       configSeconds,    -1,      -1,      DispatchReadConfig },
       SECONDS_PER_HOUR, -1,      -1,      CheckDispatchSessions },
       oneMinute * 5,    -1,      -1,      SendingMessagesToSession },
       oneMinute,        -1,      -1,      DrainSessionDescriptors },
       oneMinute,        -1,      -1,      ReportLateTimeMessage
    };

    // Initialize each elements last and next run.
    // The first run will be after sched for seconds.
    for (int i = 0; i < NUMBER_OF_ACTIVITIES; i++) {
        sched[i].lastrun = time(NULL);
        sched[i].nextrun = sched[i].lastrun + sched[i].frequency;
    }

    difference = sched[i].nextrun - sched[i].lastrun;

    // Run things forever
    while(1) {

        // Sleep for the shortest amount of time needed
        sleep(sleepfor);

        curTime = time(NULL);

        // See which activities need to be run on this pass.
        for (int i = 0; i < NUMBER_OF_ACTIVITIES; i++) {

            if (sched[i].nextrun <= curTime)
            {
                // This activity needs running. Call the function
                // and change the lastrun and next run values.
                sched[i].cleanupfunc();
                sched[i].lastrun = time(NULL);
                sched[i].nextrun = sched[i].lastrun + sched[i].frequency;

                // See how long until this needs to be run.
                difference = sched[i].nextrun - curTime;
            }

            // We need to set the sleepfor value to something on the
            // first pass so we have something to compare to.
            if ( == 0 || difference < sleepfor)
                sleepfor = difference;
        }

        // We need to set the sleepfor value to something on the
        // first pass so we have something to compare to. The
        // lowest time is what we'll sleep for.
```

```
124  3        if (i == 0 || difference < sleepfor)
125  2            sleepfor = difference;
126  1        }
127  1    } // while (1)

129  1    return buff;
110  1 }
```

```
1   /*
2   ** Copyright 1996, 1999 EMC Corporation
3   **
4   ** EDMDispatchSession.cc
5   **
6   ** Mission Statement: This is where all session management occurs.
7   **
8   ** Primary Data Acted On:
9   **
10  ** Compile-Time Options:
11  **
12  **     USE_SUNRPC - Compile source with sunrpc.  If
13  **                  not set, assume DCE support.
14  **
15  ** Basic idea here: Module for session management
17  **
18  */

20  /*
21  ** The following provides an RCS id in the binary that can be located
22  ** with the what(1) utility. The intent is to keep this short.
23  */
24  #if !defined(lint)
25  static char   RCS_id [] = "@(#)$RCSfile: EDMDispatchSession.cc,v $
26                             *Revision: 1.23 $
27                             *Date: 1997/02/06 20:49:15 $";
28  #endif

30  /* #define _POSIX_SOURCE     unable to compile with this define set */
31  /* #define _XOPEN_SOURCE     unable to compile with this define set */

33  #include <salc/c_portable.h>
34  #include <sal/op_xopen.h>
35  #include <sal/inout.h>

37  #include <pthread.h>
38  #include <memory.h>
39  #include <sys/time.h>
40  #include <sys/types.h>
41  #include <rw/vstream.h>
42  #include <rw/cstring.h>

44  // Rogue Wave includes
45  #include <rw/collect.h>
46  #include <rw/vfile.h>
47  #include <rw/bintree.h>

49  #include <csc/cscomm.h>
50  #include <restore/dispatch_daemon.h>
51  #include <restore/dispatch_protocol_client.h>
52  #include <EDMSession.h>
53  #include <EDMReturnMessageApi.h>
54  #include <EDMDbHandleMgrApi.h>
55  #include <EDMDispatchSession.h>
56  #include <EDMDispatchConfig.h>
57  #include <EDMDispatchlog.h>
58  #include <EDMSession_rstsvc.h>

60  #include <RWBinaryTree>

62  static pthread_mutex_t     G_sessionTreeMtx;
63  extern ELinkHandle*        ELinkHandle;

65  static pthread_mutex_t     G_session;
```

```
66  static int maxDisconnectTime = SECONDS_PER_HOUR;   // one hour

68  /*
69  ****************************************************************
70  **
71  ** Purpose:  Lock the session mutex.
72  **
73  ** Return Codes:  None
74  **
75  ** Outputs:  None
76  **
77  ** Inputs:   None
78  **
79  ** Routine:  LockSessionMutex
80  **
81  ****************************************************************
82  */
83  static void
84  LockSessionMutex()
85  {
86     static boolean_ty first = TRUE;
87
89     if (first == TRUE)
90     {
91        first = FALSE;
92        pthread_mutex_init(&G_sessionTreeMtx, NULL);
93     }
95     pthread_mutex_lock(&G_sessionTreeMtx);
96  }
```

```
 98
 99  /***********************************************************
100  **
101  ** Routine:     UnlockSessionMutex
102  **
103  ** Inputs:      None
104  **
105  ** Outputs:     None
106  **
107  ** Return Codes:
108  **      None
109  **
110  ** Purpose:     Unlock the mutex for the session tree object
111  **
112  ***********************************************************/
113
114  static void
115  UnlockSessionMutex()
116  {
117      pthread_mutex_unlock(&G_sessionTreeMtx);
118  }
```

```
120  /***********************************************************
121  **
122  ** Routine:     InitializeSession
123  **
124  ** Inputs:      DD_initialize_args *arg - args sent via RPC for starting
125  **                                       session
126  **              struct svc_req *req - the request block from RPC
127  **
128  ** Outputs:     DD_initialize_result *res - the result structure which
129  **                                         tells whether
130  **                                         operation succeeded or failed.
131  **
132  ** Return Codes:
133  **      None
134  **
135  ** Purpose:     Initialize a session for the GUI.
136  **
137  ***********************************************************/
138  void
139  InitializeSession(IN DD_initialize_args *arg, IN struct svc_req *req,
140                    OUT DD_initialize_result *res)
141  {
142      EDMSession *session;
143      EDMSession *ret;
144      pthread_t   id;
145      time_t      t;
146
147      if (arg == NULL || req == NULL || res == NULL)
148      {
149          return;
150      }
151
152      t = time(NULL);
153
154      session = new EDMSession();
155
156      if (session == NULL)
157      {
158          res -> status = DD_SERVICE_FAILURE_NONEXEC;
159          return;
160      }
161
162      session = InitializeSession();
163
164      session -> setStartTime(t);
165
166      session -> setOperationType(arg -> service);
167
168      session -> setStatus(DD_SERVICE_STARTING);
169
170      if (arg -> username != NULL && arg -> hostname != NULL)
171      {
172          switch(arg -> service)
173          {
174              // code is commented out because we do not
175              // want to read the config for permission information
176              // at this time, it is a waste of cycles
177  #if 0
178              case DD_SERVICE_RESTORE:
180                  allowed =
```

```
181  1      };

                    DispatchCheckRestorePermission(
                        arg->hostname,
                        arg->username);

185  4          if (!allowed)
184  4          {
185  4              res->status = DD_SERVICE_FAILURE_PERMS;
186  4              delete session;
187  3              return;
188  3          }
188  3      }
           #endif

190  1          default: // Add some error message for unknown service
                    break;
            }

194  1          };
195  1      else
196  1      {
197  2          res->status = DD_SERVICE_FAILURE_NOWEXEC;
198  2          delete session;
199  2          return;
200  2      }
201  1  }

204  1  ret = (EDMSession *) g_sessionTree.insert((
                    RWCollectable *) session);

205  1  UnlockSessionMutex();

            if (ret == NULL)
            {
210  2      res->status = DD_SERVICE_FAILURE_NOWEXEC;
211  2      delete session;
212  2      return;
213  2  }

            session->getSessionID(&res->service_handle);

216  1      // Call Steve's thread
216  1      pthread_create(&id, NULL, &DDRSvc_init, (void *) session);

221  1      session->setThreadID(id);

223  1      return;
224  1  }
```

```
226
            *****************************************************
            **
            ** Routine:    SendPingMessagesToSession
            **
            ** Inputs:     None
            **
            ** Outputs:    None
            **
            ** Return Codes:
            **             None
            **
            ** Purpose:    Queue up all the ping messages to the sessions
            **             respond they should be considered dead.  If they don't
            **
            *****************************************************
            */

            void
            SendPingMessagesToSession()
            {

248  1      LockSessionMutex();

            RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                    G_sessionTree);

            EDMSession *sess;

            while ( sessionIterator != NULL &&
                    (sess = (EDMSession) (*sessionIterator)()) != NULL )
            {
                DD_client_session_id sid;
                rpc_binding_handle_t *cscb = NULL;
                int                  status;
                int                  ret;

                if (sess->getStatus() != DD_SERVICE_RUNNING)
                    continue;

                sess = getSessionID(&sid);

                ret = GetCSCHandle(&sid, &cscb);

                if ( ret == 0 || cscb == NULL || *cscb == NULL)
                    continue;

                PushResponseMessage(dp_ping_request, sid, cscb, &status);

            // through with iterator
            if (sessionIterator != NULL)
                delete sessionIterator;
            }

            UnlockSessionMutex();
            }
```

```c
282  /****************************************************************
283  **
284  ** Routine:  UpdateSessionLastReceived
285  **
286  ** Inputs:   DD_client_session_id *sessID - session that we
287  **                                          something
288  **
289  ** Outputs:  None
290  **
291  ** Return Codes:
292  **            0 on success and non-zero otherwise
293  **
294  ** Purpose:  Update the specified session with the lastest received
295  **                                          message
296  **            time.
297  **
       */
298  int
299  UpdateSessionLastReceived(DD_client_session_id *sessID)
300  {
301      time_t   last = time(NULL);
302      EDMSession *session;
303      EDMSession *ret;
304  1
305  1   session = new EDMSession();
306  1
307  1   if (session == NULL)
308  2   {
309  2       EDMDispatch_logerr(
310  2           _FILE_,  _LINE_,  LOG_ERR, SESSION_NO_MEMORY, 0,
311  2           "Failure to create a session block");
312  2
313  1       return -1;
314  1   }
315  1
316  1   session -> setSessionID(sessID);
317  1
318  1   LockSessionMutex();
319  1
320  1   ret = (EDMSession *) Q_sessionTree.find(RWCollectable *) session);
321  1
322  1   UnlockSessionMutex();
323  1
324  1   delete session;
325  1
326  1   if (ret == NULL)
327  2   {
328  2       EDMDispatch_logerr(
329  2           _FILE_,  _LINE_,  LOG_ERR, SESSION_LOOKUP_FAILED, 0,
330  2           "Failure to update session %ld:%ld received",
331  2           sessID -> high, sessID -> low);
332  2
333  1       return 1;
334  1   }
335  1
336  1   ret -> setLastReceived(last);
337  1
338  1   return 0;
339  }
```

```c
340  /****************************************************************
341  **
342  ** Routine:  UpdateSessionLastSent
343  **
344  ** Inputs:   DD_client_session_id *sessID - session that we sent us
345  **                                          something
346  **
347  ** Outputs:  None
348  **
349  ** Return Codes:
350  **            0 on success and non-zero otherwise
351  **
352  ** Purpose:  Update the specified session with the lastest sent
353  **                                          message
354  **            time.
355  **
       */
356  int
357  UpdateSessionLastSent(DD_client_session_id *sessID)
358  {
359      time_t   last = time(NULL);
360      EDMSession *session;
361      EDMSession *ret;
362  1
363  1   session = new EDMSession();
364  1
365  1   if (session == NULL)
366  2   {
367  2       EDMDispatch_logerr(
368  2           _FILE_,  _LINE_,  LOG_ERR, SESSION_NO_MEMORY, 0,
369  2           "Failure to create a session block");
370  1       return -1;
371  1   }
372  1
373  1   session -> setSessionID(sessID);
374  1
375  1   LockSessionMutex();
376  1
377  1   ret = (EDMSession *) Q_sessionTree.find(RWCollectable *) session);
378  1
379  1   UnlockSessionMutex();
380  1
381  1   delete session;
382  1
383  1   if (ret == NULL)
384  2   {
385  2       EDMDispatch_logerr(
386  2           _FILE_,  _LINE_,  LOG_ERR, SESSION_LOOKUP_FAILED, 0,
387  2           "Failure to update session %ld:%ld sent",
388  2           sessID -> high, sessID -> low);
389  1       return 1;
390  1   }
391  1
392  1   ret -> setLastSent(last);
393  1
394  1   return 0;
395  }
```

```
394    /*****************************************************
395    **
396    ** Routine:     CheckDispatchSessions
397    **
398    ** Inputs:      None
399    **
400    ** Outputs:     None
401    **
402    ** Return Codes:    None
403    **
404    **
405    ** Purpose:     Look for dead sessions and kill them off
406    **
407    **
408    */
409    void
410    CheckDispatchSessions()
411    {
412        EDMDSession  *sess;
413        int          status = 0;
414        int          rec = 0;
415        time_t       currTime;
416        RWBinaryTree reaperTree;
417        RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                                                    G_sessionTree);

421        currTime = time(NULL);

423        LockSessionMutex();

425        while ( sessionIterator != NULL &&
426              (sess = (EDMDSession*) (*sessionIterator)()) != NULL ) {

428            if ( (sess->getLastReceived() &&
429                 currTime - maxDisconnectTime > sess->getLastReceived()) != 0) ||
430                 ( sess->getStatus(
431                 ) == DD_SERVICE_FAILURE_EXEC ||
432                 sess -> getStatus(
433                 ) == DD_SERVICE_FAILURE_NONEXEC || sess -> getStatus(
434                 ) == DD_SERVICE_FAILURE_PERMS) ) {

436                // Insert it into the reaper tree
                   // (void) reaperTree.insert(sess);
438                reaperTree.insert(sess);
439            }
440        }

442        // through with iterator
        if (sessionIterator != NULL)
444            delete sessionIterator;

        // If the reaper tree has something in it then use those entries
                                                    //       to remove
446        if (reaperTree.entries() > 0) {

448            UnlockSessionMutex();

450            sessionIterator = new RWBinaryTreeIterator(reaperTree);
```

```
452        while ( sessionIterator != NULL &&
453              (sess = (EDMDSession*) (*sessionIterator)()) != NULL ) {
454
456            sess -> getSessionID(&sessID);

458            ret = removeSession(&sessID, &status);

460            if (ret != 0) {
461                EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, 0, 0,
462                    "Failure to remove session %ld:%ld",
463                    sessID.high, sessID.low);
464                continue;
465            }

467            EDMDispatch_logent(
468                "removing session %ld:%ld",
469                sessID.high, sessID.low);

471            currTime - maxDisconnectTime > sess->getLastReceived(),

473            if ( ret != 0) {
474                ret = deleteandSet(&sessID, &ELinkHandle, &status);

476                EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO, 0, 0,
477                    "Haven't received anything since %ld. Current %ld",
478                    sessID.high, sessID.low);

480            }
481            else {
482
483            }

485            // through with iterator
            if (sessionIterator != NULL)
487                delete sessionIterator;

489        }

491        reaperTree.clear();
492    }
493    }
```

```c
495  /***********************************************************
496  **
497  ** Routine:     DrainSessionDescriptors
498  **
499  ** Inputs:      None
500  **
501  ** Outputs:     None
502  **
503  ** Return Codes:
504  **              None
505  **
506  ** Purpose:     Drain whatever data is on stdout and stderr for sessions.
507  **
508  */
509  ***********************************************************/
510
511  void
512  DrainSessionDescriptors()
513  {
514      int     hout = 0, herr = 0, status = 0;
515      int     selret = 0;
516      char    buff[1024];
517      struct  timeval timetowait = {
518          0,
519          0
520      };
521      fd_set  stdoutSet;
522      fd_set  stderrSet;
523
524      getStdoutSet(&stdoutSet, &hout, &status);
525
526      if ( (selret = select(
527              hout + 1, &stdoutSet, NULL, NULL, &timetowait)) >= 0 )
528      {
529          for (i = 0; i < hout; i++)
530          {
531              if (FD_ISSET(i, &stdoutSet))
532              {
533                  while (read(i, buff, 1024) > 0);
534              }
535          }
536      }
537
538      getStderrSet(&stderrSet, &herr, &status);
539
540      if ( (selret = select(
541              herr + 1, &stderrSet, NULL, NULL, &timetowait)) >= 0 )
542      {
543          for (i = 0; i < herr; i++)
544          {
545              if (FD_ISSET(i, &stderrSet))
546              {
547                  while (read(i, buff, 1024) > 0);
548              }
549          }
550      }
551  }
```

```c
552  /***********************************************************
553  **
554  ** Routine:     GetSessionStatus
555  **
556  ** Inputs:      DD_client_session_id *ssid - session ID to check the
557  **                                           status of
558  **
559  ** Outputs:     int *status - status of the function call
560  **              int *s_status - session status
561  **
562  ** Return Codes:
563  **              0 if successful and non-zero otherwise
564  **
565  ** Purpose:     Get status on the session.
566  **
567  */
568  ***********************************************************/
569  int
570  GetSessionStatus(
571      DD_client_session_id *ssid, int *s_status, int *status)
572  {
573      EDMSession  *sess;
574      EDMSession  *ret;
575
576      if (status == NULL)
577      {
578          return -1;
579      }
580
581      if ( ssid == NULL || s_status == NULL)
582      {
583          *status = SESSION_BAD_ARGS;
584          return -1;
585      }
586
587      sess = new EDMSession();
588
589      if (sess == NULL)
590      {
591          EDMDispatch_logmsg(
592              __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
593              "Failure to create a session block");
594          return -1;
595      }
596
597      LockSessionMutex();
598
599      sess -> setSessionID(ssid);
600
601      ret = (EDMSession *) G.sessionTree.find((RWCollectable *) sess);
602
603      UnlockSessionMutex();
604
605      delete sess;
606
607      if (ret == NULL)
608      {
609          EDMDispatch_logmsg(
                 __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                 "Failure to lookup session %ld.%ld");
```

```
610 2              "status = SESSION_LOOKUP_FAILED,
                   ssid -> high, ssid -> low);
611 2              return -1;
612 1          }
613 1
615 1          *s_status = ret -> getStatus();
617 1          return 0;
618      }
```

```
621     /*****************************************************************
622     **
623     **  Routine:   GetDispatchStatus
624     **
625     **  Inputs:    DD_getservicestatus_args *arg - session ID to check the
626     **                                             status of
627     **
628     **  Outputs:   DD_getservicestatus_result *res - the result structure
629     **                                               whether operation succeeded or failed
630     **
631     **  Return Codes:
632     **             None
633     **
634     **  Purpose:   Get status on the starting session.
635     **
637     *****************************************************************/
638     void
639     GetDispatchStatus (IN DD_getservicestatus_args *arg,
640                        OUT DD_getservicestatus_result *res)
641     {
642         EDMSession    *sess;
643         EDMSession    *ret;
644         static char buff[CONNECT_HANDLE_SIZE];
645
647 1       sess = new EDMSession();
648 1
649 2       if (sess == NULL)
650 2       {  // Give an error
651 2           EDMDispatch_logerr(
652               _FILE_,    _LINE_, SESSION_NO_MEMORY, 0,
654               "failure to create a session block");
656           return;
658       }
660       sess -> getSessionID(&arg -> service_handle);
662       LockSessionMutex();
664       ret = (EDMSession *) g_sessionTree.find(RWCollectable *) sess);
666       UnlockSessionMutex();
667       delete sess;
670 1       if (ret == NULL)
672 1       {
674 1           EDMDispatch_logerr(
676               _FILE_,   _LINE_, SESSION_LOOKUP_FAILED, 0,
                  "failure to lookup session %ld:%ld",
                  arg -> service_handle.high,
                  arg -> service_handle.low);

              res -> status = DD_SERVICE_FAILURE_NONEXEC;
              return;
          }

          res -> status = ret -> getStatus();

          memset(buff, 0, sizeof(buff));
```

```
678 1        if (res -> status == DD_SERVICE_RUNNING)
679 2            res -> handle.handle_val = (char *) ret -> getConnectionHandle(
680 2            );
681 1        else
682 2            res -> handle.handle_val = (char *) buff;
683 2
684 2        res -> handle.handle_len = CONNECT_HANDLE_SIZE;
685 2    }
686 2
687 1 }
688 1
```

```
690  **********************************************************
691  **
692  ** Routine:      GetDispatchInfo
693  **
694  ** Inputs:       DD_getservicestatus_args *arg - session ID to check the
695  **                                               status of
696  **
697  ** Outputs:      SessionBlock *res - the information regarding the
698  **                                   specified session
699  **
700  ** Return Codes:
701  **               None
702  **
703  ** Purpose:      Get status on all the sessions.
704  **
705  **********************************************************
706  */

707  void
708  GetDispatchInfo(IN DD_getservicestatus_args *arg,
                     OUT SessionBlock *res)
709  {
710 1    EDMSession     *sess;
711 1    EDMSession     *ret;
712 1    SessionInfo    *sinfo, *slast;
713 1    static char     buff[CONNECT_HANDLE_SIZE];
714
715 1    LockSessionMutex();
716
717 1    if (arg -> service.high != 0 && arg -> service.handle.
718                                     low != 0)
719 2    {
720 2        // Looking for a single session. Do a find.
721 2        sess = new EDMSession();
722 2        if (sess == NULL)
723 3        {
724 3            // Give an error
725 4            EDMDispatch_logent(
                     _FILE_, _LINE_, LOG_ERR, SESSION_NO_MEMORY, 0,
                     "failure to create a session block");
727 4
728 3            UnlockSessionMutex();
729 3            return;
730 2        }
731 2        sess -> setSessionID(arg -> service.handle);
732 2        ret = (EDMSession *) g_sessionFree.find(sess);
733
734 2        delete sess;
735
736 2        if (ret == NULL)
737 3        {
738 3            EDMDispatch_logent(
739                  _FILE_, _LINE_, LOG_ERR, SESSION_LOOKUP_FAILED,
740                  arg -> service.handle.high,
741                  arg -> service.handle.low);
742 3            UnlockSessionMutex();
743 3            return;
744 2        }
745 2        res -> totalsessions = 1;
746 2
```

```
750  2      if ( res -> sess == NULL )
751  1      {
752  2          EDMDispatch_logent(
753  3              __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
754  3              "Failure to allocate session info block");
756  3
758  2          UnlockSessionMutex();
759  2          return;
760  2      }
761  1
762  2      sinfo = res -> sess;
763  3
764  2      ret -> getSessionID(&sinfo -> service_handle);
765  2      sinfo -> status = ret -> getStatus();
766  2      sinfo -> jobstarttime = ret -> getStartTime();
767  2      sinfo -> operation_type = ret -> getOperationType();
768  2      sinfo -> lastSent = ret -> getLastSent();
769  2      sinfo -> lastReceived = ret -> getLastReceived();
771  2      }
773  2      else
774  1      {
775  2          res -> sess = (SessionInfo *) calloc(1, sizeof(SessionInfo));
777  2
778  2          if (res -> sess == NULL)
779  1          {
781  2              EDMDispatch_logent(
783  3                  __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
785  3                  "Failure to allocate session info
787  3                  block");
788  2
789  1              UnlockSessionMutex();
790  2              return;
791  2          }
792  3
793  4          sinfo = res -> sess;
794  4
795  4      }
798  2
799  2      res -> totalsessions = 0;
800  4
801  4      RWBinaryTreeIterator *sessionIterator = new
802                  RWBinaryTreeIterator(g_sessionTree);
803  3
803  4      boolean_t addnext = FALSE;
804  4
805  3      while ( sessionIterator != NULL && (ret = (EDMSession*)
                  *sessionIterator)() != NULL )
     3      {
     3          int         status;
     3
     3          if (addnext)
     3          {
     3              sinfo -> next = (SessionInfo *) calloc(1, sizeof(
                        SessionInfo));
     3
     3              if (sinfo -> next == NULL)
     3              {
     3                  break;
     4              }
     4
     4              sinfo = sinfo -> next;
     5          }
```

```
806  3          sinfo -> operation_type = ret -> getOperationType();
807  3          sinfo -> lastSent = ret -> getLastSent();
808  3          sinfo -> lastReceived = ret -> getLastReceived();
810  3
813  3          ret -> getSessionID(
816  3              &sinfo -> service_handle, &sinfo -> outhandle,
817  2              &sinfo -> erxhandle, &status);
818  3
819  2          res -> totalsessions++;
820  2
821  3          sinfo -> next = NULL;
822  2          addnext = TRUE;
823  2      }
824  2
825  1      // through with iterator
827  1      if (sessionIterator != NULL)
828            delete sessionIterator;
           }

           UnlockSessionMutex();
```

```
830   /*
831   **
832   ** Routine:  removeSession
833   **
834   ** Inputs:
835   **
836   ** Outputs:
837   **
838   ** Return Codes:
839   **     None
840   **
841   ** Purpose: Remove the active session object between the GUI and the
842   **          Service.
843   **
844   */

846   int
847   removeSession(IN DD_client_session_id *sess_id,
                    OUT int *status)
848   {
849       EDMSession *sess;
850       EDMSession *ret;
851
852       if (status == NULL)
853           return -1;
854
855       if (sess_id == NULL)
856       {
857           *status = SESSION_BAD_ARGS;
858           return -1;
859       }
860
861       *status = 0;
862       if (g_sessionTree.isEmpty())
863       {
864           EDMDispatch_logent(
                   __FILE__, __LINE__, LOG_ERR, SESSION_LIST_EMPTY, 0,
                   "No sessions in list.");
865           *status = SESSION_LIST_EMPTY;
866           return -1;
867       }
868
869       sess = new EDMSession();
870       if (sess == NULL)
871       {
872           EDMDispatch_logent(
                   __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                   "Failure to create a session block");
874           *status = SESSION_NO_MEMORY;
876           return -1;
877       }
878
879       sess -> setSessionID(sess_id);
880
881       LockSessionMutex();
884       ret = (EDMSession *) g_sessionTree.remove(sess);
```

```
890   1   UnlockSessionMutex();
892   1   if (ret == NULL)
893   2   {
894   2       EDMDispatch_logent(
                  __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                  "Failure to remove session <%ld:%ld>",
                  sess_id -> high, sess_id -> low);
895   2       *status = SESSION_LOOKUP_FAILED;
899   2       return -1;
900   1   }
902   1   delete ret;
903   1   delete sess;
905   1   return 0;
906       }
```

```c
 1   /*
 2    ** Copyright 1996, 1998 BMC Corporation
 3    */
 5   /*
 6    ** Mission Statement: This is the logging wrapper around esl logging.
 7    **
 8    ** Primary Data Acted On:
 9    **
10    ** Compile-Time Options:
11    **
12    ** Basic idea here: Module for logging
13    */
14
15   #endif
16
17   /*
18    ** The following provides an RCS id in the binary that can be located
19    ** with the what(1) utility. The intent is to keep this short.
20    */
21   #if !defined(lint)
22   static char  RCS_id [] = "@(#)$RCSfile: bamlogging.c,v $ "
23                            "$Revision: 1.23 $ "
24                            "$Date: 1997/02/06 20:49:15 $";
25   #endif
27   #include <aeslc_portable.h>
28   #include <esl/ep_xopen.h>
29   #include <esl/input.h>
31   #include <stdio.h>
32   #include <pthread.h>
33   #include <stdarg.h>
34   #include <string.h>
36   #include <logging/logging.h>
37   #include <EDMDispatchLog.h>
38   #include <EDMDispatch.h>
40   /*
41    ** Routine: EDMDispatch_logent
42    **
43    ** Purpose:
44    **     Inputs:
45    **         file        - c source file name
46    **         line number - line number in c source file
47    **         priority    - esl logent priority
48    **         message #   - esl logent message number
49    **         errno       - errno option
50    **         msg format  - format for printing.
51    **                       ie
52    **                       "text %s %d ..."
53    **
54    **     Outputs:
55    **         None
56    **
57    **     Return Codes:
58    **
59    **     All messages issued from the bamd should call this
60    **         so that they can be consistently formatted. It is
61    **         anticipated that this routine will also be used for
62    **         statistics gathering on messages and perhaps even
63    **         debugging.
64    **
65    **     Calls esl_logent for output.
66    **
67    **     variable args - variable arguments for printing
68    **
69    ** Intended caller: internal only.
```

```c
      /************************************************/
      void
      EDMDispatch_logent( IN char   *filename,
                          IN int     linenum,
                          IN int     priority,     /* esl_logent priority */
                          IN int     msg_no,       /* esl_logent message number */
                          IN int     err_no,       /* errno */
                          IN char   *msg);         /* msg format for sprintf... */
      /************************************************/

 64   #define EDMDispatch_MSGBUFF 2048
 66   {
 69       char    tmpbuff[ EDMDispatch_MSGBUFF ];
 71       char    msgbuff[ EDMDispatch_MSGBUFF ];
 72       va_list vap;
 73       static pthread_mutex_t  log_mutex = PTHREAD_MUTEX_INITIALIZER;
 74       static boolean_ty       first = TRUE;
 76
 78       if (first)
 80       {
 81           first=FALSE;
 82           pthread_mutex_init(&log_mutex, NULL);
 83       }
 84
 86       if ( msg == NULL || filename == NULL )
 87           return;
 88
 89       /* Setup variable argument list processing */
 90       va_start(vap, msg);
 91
 93       /* Build caller message with all arguments. */
 95       (void) vsprintf( tmpbuff, msg, vap );
 97
 99       if ( err_no > 0 )
100       {
103           char *str = strerror(err_no);
105
107           if (str == NULL)
109               str = "Unknown Error";
111
112           (void) sprintf(
114                   msgbuff,
115                   "(%s-%d) %s <%s>",
116                   filename,
118                   linenum,
119                   tmpbuff,
120                   str,
121                   err_no );
122       }
123       else
124       {
              (void) sprintf(
                      msgbuff,
                      "(%s-%d) %s",
                      filename,
                      linenum,
                      tmpbuff );
      }
```

```
126 1                              linenum,
127 1                              tmpbuff );

129 1     /*
130 1      * Use a mutex lock because esl_logent is NOT thread safe.
131 1      */
132 1     pthread_mutex_lock(&log_mutex);

134 1     (void) esl_logent ( priority, EB, EDMDISPATCH, msg_no, msgbuf );

136 1     pthread_mutex_unlock(&log_mutex);
137 1 } /* End of EDMDispatch_logent() */
```

```
  1   /*
  2    */
  3
  6   /*
  7    * Copyright 1996, 1997 EMC Corporation
  8    *
  9    * EDMDispatchService.c
 10    *
 11    * Primary Data Acted On:
 12    *
 13    * Mission Statement:  RPC entry points.
 14    *
 15    * Compile-Time Options:
 16    *
 17    * Basic Idea here:
 18    *
 19    */

      #if !defined(lint)
      static char    RCS_id [] =  "@(#)$RCSfile: EDMDispatchService.c,v $ "
                                  "$Revision: 1.0 $ "
 22                               "$Date: 1997/02/06 20:49:15 $ ";
 21   #endif

 20   #include <eslrc_portable.h>
      #include <esl/1/nout.h>

 24   #include <logging/logging.h>
 25   #include <csc/cscomm.h>

 28   #include <restore/csc_EDMDispatch.h>
      #include <restore/dispatch_daemon.h>

 30   #include <EDMDispatchlog.h>
 31   #include <EDMDispatchSession.h>

 34   /*
      * These are all the rpc entry points for the dispatch daemon.
 36   * The dispatch daemon is multi-threaded and it is the main thread
 37   * which handles all incoming RPC. ONC RPC is single threaded
 39   * so each call blocks other RPC calls. This provides us some
 40   * safety in the way we handle our data and limits our exposure
 41   * to unexpected multithreading problems.
 43   */
 44
 46   static void FreeSessionInfo(SessionInfo *);
 47
 48   /*
 49   **
 50   ** Routine:   dd_initialize_1
 51   **
 52   ** Inputs:    DD_initialize_args * - args for the restore initialize
 53   **                                   call
 54   ** Outputs:   None
 55   **
 56   ** Return Codes:
 57   **            DD_initialize_result * - result of init function call
 58   **
 59   ** Purpose:   Function to create a restore session.
 60   **
 61   ** Intended caller:    Internal Only.
      **
      */
```

```
 64   DD_initialize_result *
 65   dd_initialize_1_svc(
 66       IN DD_initialize_args *arg, IN struct svc_req *req )
 67   {
          static DD_initialize_result argzz;

 70        InitializeSession(arg, req, &argzz);

 71        return &argzz;
      }
```

```
73   /* *******************************************************
74    **
75    ** Routine:    dd_getservicestatus_1
76    **
77    ** Inputs:     dd_getservicestatus_args * - args for the
78                                                getservicestatus call
79    **
80    ** Outputs:    None
81    **
82    ** Return Codes:
83    **    DD_getservicestatus_result * - result of status function
84                                         call
85    **
86    ** Intended caller:  Internal Only.
87    **
88    */
89   DD_getservicestatus_result *
90   dd_getservicestatus_1_svc(
91       IN DD_getservicestatus_args *arg, IN struct svc_req *req )
92   {
93       static DD_getservicestatus_result argzz;
94
95       GetDispatchStatus(arg, &argzz);
96
97       return &argzz;
98   }
```

```
100  /* *******************************************************
101   **
102   ** Routine:    dd_getsessioninfo_1
103   **
104   ** Inputs:     DD_getsessioninfo_args * - args for the getsessioninfo
105                                             call
106   **
107   ** Outputs:    None
108   **
109   ** Return Codes:
110   **    DD_getsessioninfo_result * - result of session info call
111   **
112   **
113   ** Intended caller:  Internal Only.
114   **
115   ** Purpose:  Function to get information on all sessions.
117  SessionBlock *
118  dd_getsessioninfo_1_svc(
119      IN DD_getsessioninfo_args *arg, IN struct svc_req *req )
120  {
121      static SessionBlock argzz;
122      static boolean_t first = TRUE;
123
124      if (first)
125          memset(&argzz, 0, sizeof(argzz));
126      else
127      {
128          FreeSessionInfo(argzz.sess);
129          argzz.sess = NULL;
130      }
132      first = FALSE;
134      GetDispatchInfo(arg, &argzz);
136      return &argzz;
137  }
```

```c
139  /*
140   **
141   ** Routine:  FreeSessionInfo
142   **
143   ** Inputs:   SessionInfo * - arg to free
144   **
145   ** Outputs:  None
146   **
147   ** Return Codes:
148   **           None
149   **
150   ** Purpose:  Function to free all SessionInfo structures in a list.
151   **
152   ** Intended caller: Internal Only.
153   ***********************************/

155  static void FreeSessionInfo(SessionInfo *sess)
156  {
157      if (sess == NULL)
158          return;

160      if (sess -> next != NULL)
161          FreeSessionInfo(sess -> next);

163      free(sess);
164  }
```